42. BWInf Runde 2 - Aufgabe 3

Inhaltsverzeichnis

[Aufgabenstellung 2](#_Toc162525355)

[Beispiel zur Aufgabenstellung 2](#_Toc162525356)

[Lösungsidee 2](#_Toc162525357)

[Umsetzung 6](#_Toc162525358)

[Klassendiagramm 6](#_Toc162525359)

[Modell 8](#_Toc162525360)

[Optimierung 10](#_Toc162525361)

[Lauf 10](#_Toc162525362)

[Beobachter 11](#_Toc162525363)

[Evaluation 13](#_Toc162525364)

[Durchführung: Erstellung der Besiedlungspläne 13](#_Toc162525365)

[Vorgegebene Siedlungsgebiete 14](#_Toc162525366)

[Zusätzliches Siedlungsgebiet 18](#_Toc162525367)

[Beispiele 20](#_Toc162525368)

[Optimierer 20](#_Toc162525369)

[Lauf 22](#_Toc162525370)

[Quellcode 25](#_Toc162525371)

[Gebiet: ist\_drin 25](#_Toc162525372)

[Besiedlungsplan: auswerte\_ort\_abstand 26](#_Toc162525373)

[GierigOptimierer: tue 27](#_Toc162525374)

# Aufgabenstellung

In der Aufgabe wird nach einem Besiedlungsplan gesucht, der eine maximale Anzahl von Ortschaften im Gebiet platziert unter Berücksichtigung der Mindestabstände. Dazu soll es **genau ein** Gesundheitszentrum geben, da "... genügend Geld zur Verfügung [stehe], um ein Gesundheitszentrum zu bauen".

In der Realität werden hingegen bei Investitionsentscheidungen der **Nutzen** und die **Kosten** gegenübergestellt. In diesem Fall ist also die Frage, wie viele weitere Ortschaften in den Besiedlungsplan korrekt aufgenommen werden können, wenn ein Gesundheitszentrum **zusätzlich** gebaut wird. Diese Frage muss für jede (sinnvolle) Zahl von Gesundheitszentren beantwortet werden. Darauf basierend kann der Planer den optimalen Besiedlungsplan wählen und umsetzen.

## Beispiel zur Aufgabenstellung

Für *ein gegebenes Gebiet* und *den Mindestabständen zwischen Ortschaften* werden folgende optimale Pläne *abhängig von der Zahl der Gesundheitszentren* gefunden:

| **# Gesundheitszentren** | **# Ortschaften** |
| --- | --- |
| 0 | 43 |
| 1 | 107 |
| 2 | 125 |
| 3 | 127 |

Tabelle 1: Beispielhafter Vergleich der Anzahl an Ortschaften zur Anzahl der Gesundheitszentren für ein gegebenes Gebiet

Das erste Gesundheitszentrum lohnt sich also, wenn seine Kosten den Nutzen zusätzlicher Ortschaften unterschreiten. Beim zweiten müssten die Kosten unter dem Nutzen zusätzlicher Ortschaften liegen. Und das dritte Gesundheitszentrum erlaubt nur weitere Ortschaften, seine Kosten übersteigen hier also wohl den Nutzen dieser beiden Ortschaften.

Während die ursprüngliche Aufgabestellung also nur eine Zeile der Tabelle fordert, wird die Aufgabenstellung so erweitert, dass nach der gesamten Tabelle gesucht wird.

# Lösungsidee

Der Besiedlungsplan wird *schrittweise* um *je eine Ortschaft* erweitert. Bei jedem Schritt werden die *Abstandsbedingungen* überprüft und der Plan so *optimiert*, dass diese Bedingungen alle erfüllt sind. Wenn dies auch nach mehrmaligen Versuchen nicht erfolgreich ist, wird *ein Gesundheitszentrum* hinzugefügt.

Die Kernpunkte des Ansatzes sind also wie folgt:

* **Optimierer**: Für einen gegebenen Plan werden Ortschaften und Zentren so lange innerhalb des Gebiets verschoben, bis alle Abstandsbedingungen erfüllt sind. Dazu werden in jeder Iteration Plankandidaten durch Punktverschiebung generiert. Diese Kandidaten werden anhand ihres *Loss* beurteilt, der sich aus der Summe fehlender Abstände zwischen Punktpaaren berechnet. Bei der gierigen Optimierung wird dann der Kandidat mit dem geringsten *Loss* für die nächste Iteration ausgewählt. Liegt der *Loss* bei null, ist eine *zulässige* Lösung gefunden und die Optimierung beendet. Ansonsten wird die Optimierung mit einer *unzulässigen* Lösung *abgebrochen*, wenn die Zahl der Iterationen einen Schwellwert überschreitet oder seid zu vielen Iteration keine Verminderung des Loss eingetreten ist.
* **Lauf**: Startend von einem *leeren* Gebiet wird eine Ortschaft nach der nächsten *zufällig innerhalb* des Gebiets platziert. Wird der Plan dadurch *unzulässig*, werden die Planpunkte durch die *Optimierung* so *angepasst*, dass der Plan *zulässig* wird. Ist dies nicht möglich, wurde die Ortschaft eventuell an einer Engstelle platziert. Als eine Art *Backtracking* werden daher weitere *zufällige Platzierungen* für die neue Ortschaft ausprobiert (Lauf-Iteration). Scheitern diese Versuche, ist davon auszugehen, dass das Gebiet keine weitere Ortschaft aufnehmen kann. Daher wird ein *Gesundheitszentrum hinzugefügt* und mit der zufälligen Platzierung der neuen Ortschaft erneut begonnen. Der Lauf endet, wenn bereits so viele Zentren auf dem Plan sind, dass das gesamte Gebiet von ihnen praktisch abgedeckt werden könnte (in den Beispieldaten bei ca. drei Zentren der Fall).

A diagram of a diagram

Description automatically generated

Abbildung 1: Ablauflogik des Lösungsansatzes

Für die Komplexitätsbetrachtung sind zwei Aspekte von Interesse. Hierfür sei die Zahl der *Ortschaften* ***n***, der *Gesundheitszentren* ***z*** und diejenige der *Eckpunkte des Gebietspolygons* ***g***:

* **Loss-Berechnung**: Vorab ist in *O*(*n* ⋅ *z*) zu überprüfen, welche Orte von Zentren geschützt sind. Darauf basierend lässt sich für Ortspaare deren Abstand berechnen und auf Zulässigkeit überprüfen. Wird dies für *alle* Ortspaare gemacht, würde dies *O(n*2) an Aufwand bedeuten. Um das zu vermeiden, werden die Ortschaften in *Planquadrate* aufgeteilt, deren Seitenlänge der *strengsten* Abstandsregel entspricht (in der Aufgabenstellung also *20 km*). Dann müssen nur für Ortspaare *benachbarter* Planquadrate die Abstände berechnet und überprüft werden (auch diagonale Nachbarschaft berücksichtigt). Da sowohl die Zahl der Ortschaften pro Planquadrat als auch die Zahl benachbarter Planquadrate *konstant* ist, fällt nur der *assoziative Zugriff* auf Planquadrate mit *O(log n)* bei der Komplexitätsbetrachtung in Gewicht. Dies führt zum Aufwand von *O*(*n log* *n*) für die Ortspaare. Zusammen ergibt das *O(n ⋅ (z + log n))*.
* **Gebiet-Überprüfung**: Mit dem *Punkt-in-Polygon* Algorithmus erfordert die Überprüfung eines Punkts *Θ(g)*. Jeder Punkt wird so lange von seinem Ausgangspunkt verschoben, bis er innerhalb des Gebiets liegt. Da hierbei die Punktabstände nicht berücksichtigt werden, gibt es abhängig von der Gebietsform einen konstanten Faktor, der sich aus der Wahrscheinlichkeit vergeblicher Versuche ergibt. Dieser ist für die Komplexitätsklasse unerheblich. Da nicht immer alle Punkte bewegt werden und für unbewegte Punkte die Gebiet-Überprüfung schon erfolgt ist, liegt der Aufwand für einen Plankandidaten somit bei *O(g ⋅ (n + z))*.

Daraus ergibt sich folgende Komplexität für die Elemente der Ablauflogik:

* *Optimierer-Iteration:* Die Zahl der Kandidaten ist durch Konstanten festgelegt. Damit ergibt sich die Komplexität direkt aus der Kombination der *Loss-Berechnung* und der *Gebiet-Überprüfung*, also: *O(n log n + n ⋅ (g + z) + g ⋅ z)*
* *Lauf-Iteration:* Auch hier ist durch Konstanten festgelegt, wie viele Iterationen maximal vom Optimierer durchlaufen werden und wie viele Platzierungen der neuen Ortschaft versucht werden dürfen. Die Komplexität ist also dieselbe wie die einer *Optimierer-Iteration*.
* *Gesamter Lauf:* Der Besiedlungsplan durch schrittweiser Hinzunahme von Ortschaften aufgebaut. Wenn *n* die Ortschaften der abschließenden Lösung darstellt sind also *Θ(n)* Schritte obiger Komplexität durchzuführen. Dabei ist unerheblich, dass *n* anfangs kleiner ist, da dies im Schnitt nur einen konstanten Faktor ausmacht. Für die Komplexität des gesamten Laufs gilt also: *O(n2 log n + n2 ⋅ (g + z) + n ⋅ g ⋅ z)*

Prinzipiell gibt es auch alternative Lösungsansätze, die hier nicht weiterverfolgt werden. Dies liegt an Folgendem:

* **Analytische** bzw. **konstruierende** Ansätze sind wenig vielversprechend. Dies liegt an der Form der Siedlungsgebiete, die nicht einmal *konvex* sind. Eine Ausnahme ist das quadratische Siedlungsgebiet von *Siedler3*. Aber selbst hier lässt sich nur dann beweisbar optimal konstruieren, wenn kein Gesundheitszentrum verwendet wird.
* Die Verwendung von **Loss-Gradienten** *("Abstoßung")* bei der Punktverschiebung ist schwierig, da dann auch der Verbleib / die Rückkehr in die Gebietsgrenzen *("Anziehung")* mathematisch erfasst werden müsste. Daher wird auf sie verzichtet. Dennoch ist die Optimierung *zielgerichtet*, da der stetige Loss für die Kandidatenauswahl verwendet wird. Insofern bekommt der Optimierer selbst bei unzulässigen Planen ein Hinweis darauf, welcher Kandidat die Punkte am meisten in die richtige Richtung verschoben hat.
* Die vorgestellte Methode zur Planwahl einer Optimierer-Iteration ist *gierig*. Dies würde ein Problem darstellen und potenziell zu *lokalen Optima* führen, wenn Folgepläne anhand der *Gradienten* deterministisch bestimmt würden. Dies aber nicht der Fall ist, da Planpunkte *zufällig* bewegt werden. Daher kann auf zusätzliche Methoden der Planwahl (z.B. *Annealing*) verzichtet werden.

# Umsetzung

Das Programm ist mit *Python 3.11.8* geschrieben. Im Folgenden wird zunächst das Klassendiagramm als Übersicht gezeigt. Im Anschluss werden die Python-Klassen nach Paketen aufgeteilt und genauer betrachtet. Dabei wird nur auf Aspekte eingegangen, die im Klassendiagramm nicht klar erkennbar sind.

## Klassendiagramm

Um die Abbildung zu vereinfachen, ist im UML Klassendiagramm Folgendes weggelassen:

* Hole-Methoden: durch die Attributauflistung nicht zwingend erforderlich.
* Inhalt des Files "types.py", da dieser nur TPunkt (siehe unten) enthält.
* Override von Standard-Methoden (lt, str): im Modell standardmäßig durchgeführt
* Konstruktoren: nur aufgelistet, wenn ihre Parameter sich maßgeblich von den Attributen unterscheiden.
* konstante Variablen, wie zum Beispiel im *Plotter* die Farben der gültigen Ortschaften.

Außerdem wird für den Typ einen Punkt tuple[float, float] ein spezifischer Typ TPunkt verwendet. Mit Punkt sind alle Modellpunkte gemeint, also Ortschaften, Gesundheitszentren und Polygonecken.

Die Sichtbarkeit +/- vor den Attributen bedeutet, dass das Attribut privat ist, also von außen nicht veränderbar ist, jedoch auf den Inhalt des Attributs durch die Hole-Methoden zugegriffen werden kann.

Die Unterklassen von Beobachter werden in einem separaten Schaubild dargestellt.

A diagram of a computer

Description automatically generated with medium confidence

Abbildung 2: Klassendiagramm

Die verschiedenen Unterklassen von Beobachter sind für die Verfolgung und Protokollierung des Laufs und Optimierers zuständig. Sie sind im folgenden Schaubild dargestellt.

A screenshot of a computer

Description automatically generated

Abbildung 3: Klassendiagramm der Beobachter

## Modell

### Parameter

Enthält die vorgegebenen Mindestabstände zwischen den Ortschaften und auch den Schutzradius der Gesundheitszentren.

### Gebiet

Objektattribute:

* linienzug, die Liste der Eckpunkte des Polygons. Der erste und letzte Punkt sind dieselben.
* eckpunkte, die Menge der Eckpunkte. Dies ermöglicht Mengenoperationen und assoziativen Zugriff von Verwendern (z.B. die *Trafo* von *Plotter*).
* name, der Name der Datei ohne Suffix, die das Gebiet definiert. Dies wird für die Protokollierung benötigt (z.B. Verzeichnis *siedler1*).

Konstruktor:

Basiert auf dem Linienzug und dem Namen des Gebiets für den Fall, dass es aus einer Datei erstellt wird.

Methoden:

* *(statisch)* von\_datei
* Entnimmt die Eckpunkte der angegebenen Datei und schließt den Linienzug durch Hinzufügen des ersten Punkts am Ende.
* ist\_drin
* Überprüft, ob sich ein Punkt im Gebiet befindet (Punkt-in-Polygon-Test).
* Dazu ist die Strahlmethode implementiert: Alle Schnittpunkte des Polygons mit einem horizontalen Strahl von außen bis zum Punkt werden gesammelt. Ist ihre Zahl ungerade, so ist der Punkt drin.
* Quellcode im Anhang
* zufaelliger\_punkt
* Ermittelt zufällig einen gültigen Punkt, also ein Punkt im Gebiet.
* Dazu werden die kleinsten/größten x-/y-Werte des Gebiets bestimmt.
* Zufällige Wahl der Punkt-Koordinaten, die im Gebiet Werteintervall liegen.
* Überprüfung der Gültigkeit mit ist\_drin.
* Neuauswahl der Punkt-Koordinaten bis der Punkt gültig ist.

### Besiedlungsplan

Objektattribute:

* Werden im Konstruktor ermittelt und voneinander abgeleitet werden (z.B. loss). Damit wird ermöglicht, dass der wiederholte Zugriff auf sie effizient *in O(1)* ist.
* Sie sind privat, da sie konsistent gehalten werden müssen. Werden zum Beispiel die Koordinaten der Ortschaften von außen verändert, hätte dies auch Auswirkungen auf loss und geschuetzte\_orte.
* Lesender Zugriff wird über entsprechende hole Methoden ermöglicht. Die Mengen-Attribute werden als frozensets zurückgegeben, damit auch ihr Inhalt von außen nicht geändert werden kann.

Konstruktor:

* Wenn der Aufrufer bereits sichergestellt hat, dass alle Ortschaften und Gesundheitszentren im Gebiet sind, kann der Parameter alles\_im\_gebiet gesetzt werden. Damit unterlässt der Besiedlungsplan die Gebietsprüfungen. Dann fällt der Aufwand *O(g\*(n+z))* nicht erneut an.
  + Zum Beispiel ist dies bei Aufrufen des *Optimierers* und *Laufs* gegeben, die selbst die Punkteprüfung übernehmen.

Methoden:

* auswerte\_ort\_abstand
  + Berechnet den Abstand der Ortschafts-Paare und leitet daraus die Menge zunahe\_orte und den loss ab.
  + Um *O(n2)* zu vermeiden, wird die Menge der Ortschaften in Planquadrate aufgeteilt.
  + Quellcode im Anhang
* auswerte\_gebiet
* Der Loss wird erhöht in Abhängigkeit der Ortschaften außerhalb des Gebiets.
* Wird nur verwendet, wenn im Konstruktor alles\_im\_gebiet nicht gesetzt ist.
* berechne\_abstand
  + Der Abstand zwischen zwei Punkten wird mit dem Satz des Pythagoras berechnet.

Overrides:

* str
  + Gibt die Attribute als Text zurück.
  + Wird von Beobachtern benötigt, um die Ergebnisse festzuhalten.
* lt
  + Ermöglicht den direkten Vergleich zweier Besiedlungspläne anhand ihres Loss. Dies wird für die zielgerichtete Optimierung benötigt.
  + Basierend auf diesem „weniger als“ (**l**ower **t**han) werden die anderen Vergleichsoperatoren (z.B. >, ==) automatisch abgeleitet aufgrund der Annotation @functools.total\_ordering.

## Optimierung

### GierigOptimierer

Objektattribute:

* plan: Der Besiedlungsplan, an dem die Optimierung zu starten ist.
* beobachter: Liste von Beobachtern, die über den Fortgang der Optimierung zu informieren sind.

Methoden:

* tue
  + Führt die Optimierung entsprechend der Ablauflogik durch.
  + Generiert drei Typen von Planvarianten:
* Nur die *ungültigen* (d.h. die zu nahen) Ortschaften werden bewegt, da der *Loss* von ihnen verursacht wird. Die *gültigen* Ortschaften *bleiben* also in Position, womit insbesondere bei vielen Ortschaften die Wahrscheinlichkeit, eine verbesserte Planvariante zu finden, höher ist.
* *Genau ein* Gesundheitszentrum wird bewegt. Immer *alle oder keine* zu bewegen, entfällt als Option, da sonst bei den bereits *gut platzierten* Gesundheitszentren sehr wahrscheinlich *Lücken* entstehen würden.
* *Alle* Ortschaften und *ein* Gesundheitszentrum (wenn vorhanden) werden bewegt. Dies führt bei wenigen Ortschaften am schnellsten zu zulässigen Plänen.
  + Zurückgegeben wird außer dem besten Besiedlungsplan die Zahl der Iterationen, die für die Optimierung benötigt wurden.
  + Quellcode im Anhang
* erstelle\_kandidat
  + Für einen gegebenen Plan werden die Ortschaften und/oder die Zentren gemäß der Steuerparameter und gibt den resultierenden Plan zurück.
* bewege\_punkt
  + Bewegt einen Punkt und überprüft, dass sich das Ergebnis im Gebiet befindet. Ist das nicht der Fall, wird die Bewegung erneut mit dem Ausgangspunkt versucht. Der erhaltene Punkt im Gebiet wird als Resultat zurückgegeben.
  + Die maximale Entfernung, die ein Punkt (Ortschaft oder Zentrum) bei der Kandidatenerstellung verschoben werden kann, ist *10* nach der Tschernyschow-Norm (also *max(Δx, Δy)*).

## Lauf

### GierigLauf

Objektattribute:

* gebiet: Das Gebiet, für das die Besiedlungsplan zu erstellen sind.
* beobachter: Liste von Beobachtern, die über den Fortgang des Laufs zu informieren sind.

Konstruktor:

* Erstellt das Gebiet basierend auf den gegebenen Namen der Datei, die dessen Polygondaten enthält.

Methoden:

* tue
  + Führt den Lauf entsprechend der Ablauflogik durch.
  + Auch für das Setzen eines zusätzlichen Gesundheitszentrum gibt es ein Backtracking: Findet der Optimierer für die nächste Ortschaft keinen zulässigen Plan, wird dieses neue Gesundheitszentrum zufällig an einen anderen Punkt gesetzt. Erst dies ermöglicht, das Auffinden von Gebietsbereichen, die von einem Zentrum profitieren können.
  + Es gibt keinen Rückgabewert, da die Beobachter bereits die Darstellung und Protokollierung der Ergebnisse übernehmen.

## Beobachter

Die Beobachter haben die Aufgabe, die Lauf- und Optimierungsergebnisse zu darzustellen und zu protokollieren (in dem *output* Verzeichnis).

### Beobachter

Diese abstrakte Klasse bündelt die Definition der Methoden, die vom Lauf und Optimierer aufgerufen werden. Durch diese Entkopplung müssen diese Verwender keine Referenz zu spezifischen Beobachtern halten. Zusätzlich ermöglichen die geschützten Methoden die Wiederverwendung von Programmlogik, die von mehreren Unterklassen benötigt werden.

*(abstrakte)* Methoden:

* optimierer\_start: Aufzurufen am Anfang einer Optimierung.
  + Darauf basierend können Unterklassen ihre Protokollverzeichnisse und -dateien vorbereiten.
* optimierer\_iteration: Aufzurufen bei jeder Iteration des Optimierers.
  + Die Unterklassen stellen den aktuellen Besiedlungsplan dar (grafisch, Konsole, …) und protokollieren ihn.
* optimierer\_ende: Aufzurufen am Ende einer Optimierung.
  + Die Unterklassen schließen hier die Protokollierung der Optimierung ab.
* lauf\_start: Aufzurufen am Anfang eines Laufs.
* lauf\_iteration: Aufzurufen bei jeder Iteration des Laufs.
* lauf\_ende: Aufzurufen am Ende eines Laufs.

*(geschützte)* Methoden:

Obwohl diese zwei Methoden sehr ähnliche Operationen durchführen, können sie nicht zusammengefasst werden, ohne zusätzliche Steuerparameter hinzuzufügen (was hier vermieden wird). Stattdessen gibt es weitere private Methoden, durch die gleiche Teile der Programmlogik wiederverwendet werden können.

* vorbereite\_output\_file
  + Garantiert, dass die Output-Datei leer ist und bereit zum Schreiben von Daten ist.
* vorbereite\_output\_folder
  + Garantiert, dass die Output-Ordner leer ist und bereit zum Schreiben von Dateien ist.

*(private)* Methoden:

Diese Methoden dienen dazu, Dateien oder Ordner zu löschen beziehungsweise zu erstellen.

* garantiere\_kein\_file
  + Löscht die gegebene Datei, wenn vorhanden.
* garantiere\_leeren\_folder
  + Löscht diesen Ordner, falls er bereits existiert.
  + Kreiert danach diesen Ordner.
* garantiere\_folder
  + Konstruiert den Pfad, in dem sich der Ordner befinden soll.
  + Wenn noch kein Ordner an dem Pfad befindet, so wird er erstellt.
  + Ein existierender Ordner wird aber nicht verändert, da eventuell andere Beobachter-Unterklassen darin schon Dateien oder Ordner vorbereitet haben.

### Logger

* Schreibt nach jeder Iteration des Optimierers die wichtigsten Daten (zum Beispiel wie groß der Loss ist) in ein Textdokument.
* Schreibt nach jeder Iteration des Laufs den Besiedlungsplan (benutzt hierzu dessen String-Darstellung) in ein eigenes Textdokument.

### Stats

* Schreibt nach jeder Iteration des Optimierers die wichtigsten Daten (zum Beispiel wie viele Ortschaften zu nahe sind) in ein csv-Dokument.
* Schreibt nach jeder Iteration des Laufs drei Daten des Besiedlungsplan (Anzahl an Ortschaften, Gesundheitszentren und die benötigten Iterationen) in ein csv-Dokument.

### Plotter

Seine Aufgabe ist es, den Besiedlungsplan zu malen, damit der Fortschritt visuell/anschaulich dargestellt und protokolliert wird.

* Kreiert ein tkinter.tk Fenster und eine Zeichenfläche (Canvas) im Konstruktor für die weitere Verwendung.
* Malt nach jeder Iteration des Optimierers den Plan ins Fenster und speichert diesen auch in einem Optimierer-Ordner als eine png Datei ab.
* Malt nach jeder Iteration des Laufs den Besiedlungsplan ins Fenster und speichert diesen auch in einem Lauf-Ordner als eine png Datei ab.

*(private)* Klassen:

* Trafo
  + Nimmt die Punkte des Besiedlungsplans (Ortschaften, Zentren, Gebietseckpunkte) und berechnet basierend auf der Zeichenfläche-Größe eine Transformation ("Trafo"), welche die Punktkoordinaten so normalisiert, dass die Canvas genau vollständig ausgenutzt wird.
  + Dies wird von den Male-Methoden benutzt, um die Modell-Koordinaten der Punkte in die Koordinaten der Zeichenfläche zu übersetzen.

*(private)* Methoden:

* male
  + Kreiert die Trafo.
  + Ruft die anderen Male-Methoden auf, die für das Malen bestimmter Aspekte zuständig sind.
* male\_gebiet
  + Malt das Gebiet-Polygon.
* male\_zentren
  + Malt für jedes Gesundheitszentrum den Mittelpunkt und den Schutzbereich ein.
* male\_orte
  + Malt die Ortschaften mit einer bestimmten Farbe ein:
* Grün, wenn die Ortschaft gültig ist.
* Lila, wenn die Ortschaft zu nahe an einer anderen liegt und nicht unter Schutz des Gesundheitszentrums liegt.
* Blau, wenn die Ortschaft zu nahe an einer anderen liegt und einer der beiden Ortschaften sich unter dem Schutz eines Gesundheitszentrums befindet.
* speichere\_canvas\_zu\_file
  + speichert die Zeichenfläche (Bild des Besiedlungsplanes) ab in einer png-Datei mit Hilfe eines PIL.Image und *GhostScript*.

# Evaluation

## Durchführung: Erstellung der Besiedlungspläne

Die fünf vorgegebenen *Siedlungsgebiete* haben sehr reguläre Formen, die sich zur Untersuchung gewisse Aspekte des Ansatzes sehr gut eignen. Was allerdings fehlt, ist ein Siedlungsgebiet, das insofern *realistisch* ist, als dass es die *Unförmigkeiten* von Landesgrenzen oder Küstenverläufen abbildet. Dazu eignet sich das *griechische Festland* mit seinen *zerklüfteten Küsten* und *konkaven Elementen* hervorragend. Außerdem ist zu untersuchen, wie gut der Lösungsansatz mit der Komplexität des Gebiets und der verfügbaren Fläche *skaliert*.

Dafür wurde ein *sechstes* Gebiet erstellt und der Evaluationsreihe hinzugefügt. Seine Koordinaten sind im *korrekten* Maßstab (km) angegeben, womit die Abstandsregeln (*10 km*, *20 km*, *85 km*) besser *veranschaulicht* werden und eine viel größere Fläche als in den vorgegebenen Gebieten zu besiedeln ist (Achse von Nord-Süd und West-Ost ist knapp *600 km*). Außerdem besteht sein Polygon aus *165* Eckpunkten, was weitaus mehr als die *4-22* Eckpunkte der vorgegebenen Gebiete ist. Damit lässt sich die *Skalierbarkeit* der Polygon-Algorithmen auf die Probe stellen.

A map of greece with a blue outline

Description automatically generated

Abbildung 4: Zusätzliches Siedlungsgebiet: griechisches Festland als 165-Eck im korrekten Maßstab (km), also mit einer Achsenlänge von fast 600

Für jedes Siedlungsgebiet wurde je ein Optimierungslauf auf einem Desktop PC mit dem *AMD FX8330* Prozessor durchgeführt. Dabei wurde Standard-Python im Terminal verwendet und auf Kompilierung mit etwa *Cython* oder *PyPy* verzichtet. Die Dauer pro Lauf ist wie folgt:

| **Gebiet** | **Siedler1** | **Siedler2** | **Siedler3** | **Siedler4** | **Siedler5** | **Griechenland** |
| --- | --- | --- | --- | --- | --- | --- |
| Dauer / min | 83 | 62 | 103 | 132 | 68 | 2851 |

Tabelle 2: Benötige Zeit des Optimierungslaufs für jedes Gebiet

## Vorgegebene Siedlungsgebiete

In der Aufgabenstellung wird nach dem Zusammenhang gefragt zwischen der Zahl der verfügbaren Gesundheitszentren und der Anzahl der Ortschaften, die damit zulässig platziert werden können. Dies wird in der folgenden Tabelle 3 und der Abbildung 5 festgehalten.

| **# Gesund-heitszentren** | **# Ortschaften** | | | | |
| --- | --- | --- | --- | --- | --- |
| **Siedler1** | **Siedler2** | **Siedler3** | **Siedler4** | **Siedler5** |
| 0 | 39 | 37 | 49 | 49 | 41 |
| 1 | 109 | 81 | 158 | 132 | 128 |
| 2 | 129 | 81 | 162 | 150 | 131 |
| 3 | 132 | 83 | 163 | 152 | 133 |

Tabelle 3: Maximal erreichte Anzahl an Ortschaften nach Anzahl an Zentren für die vorgegebenen Gebieten

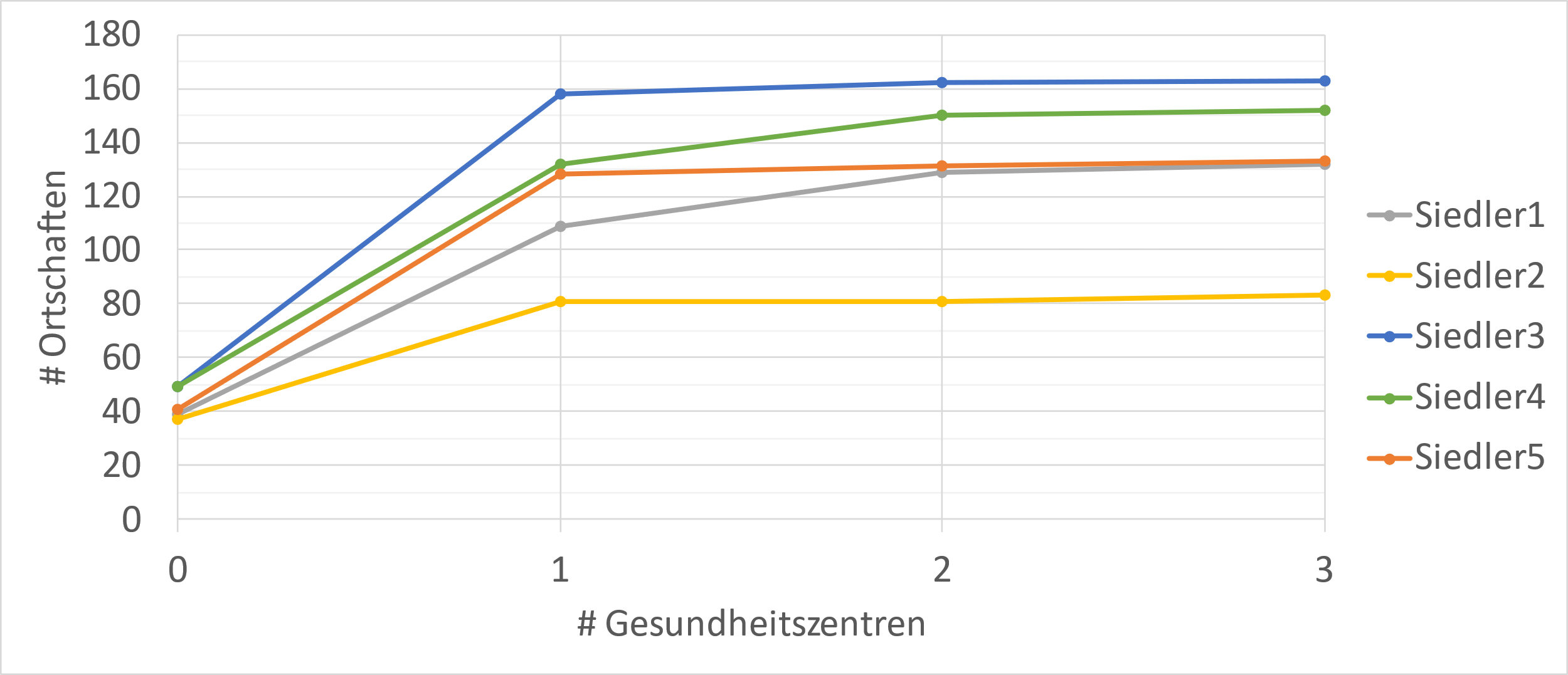


Abbildung 5: Maximal erreichte Anzahl an Ortschaften nach Anzahl an Zentren für die vorgegebenen Gebieten

Die zugehörigen Plankarten sind im Folgenden abgebildet. Sie sind auf eine *quadratische* Zeichenfläche normalisiert. Die unterschiedlichen *Zoomfaktoren* in x/y-Richtung werden z.B. bei *Siedler4* und *Siedler5* durch die *elliptischen* Schutzgebiete der Gesundheitszentren deutlich. Die Plan-Rohdaten finden sich in output/siedler\*/gierig/lauf/plandaten.

A screenshot of a computer

Description automatically generated

Abbildung 6: Plankarten für die vorgegebenen Gebiete: maximale Anzahl an Ortschaften für eine gegebene Anzahl an Zentren

Aus den Plankarten lassen sich folgende Erkenntnisse gewinnen:

* *Gesundheitszentren* werden im Zuge des Lauf in sinnvolle Positionen verschoben. So ist z.B. für *Siedler5* das erste Gesundheitszentrum *mittig* angelegt, während beim Hinzufügen eines zweiten beide Zentren sich auf beide Seiten *verteilen*. Dies beweist den Kerngedanken des Lösungsansatzes, dass auch ohne *Gradienten* nur basierend auf *Loss* eine **zielgerichtete Optimierung** stattfinden kann. Das gilt auch für bereits platzierte Ortschaften und wird aus den Optimierungsverläufen klarer deutlich. Detailliertere Beispiele aus dem Anhang:
  + *Schleichende Migration*: Das erste Gesundheitszentrum von *Siedler1* fängt am Rand an und bewegt sich dann in Richtung Mitte. Die verkürzte Darstellung des Laufes ist unter „*Beispiele/Lauf/Siedler1*“ zu sehen.
  + *Überwinden von Gebietslücken:* Auch *Lücken* im Gebiet werden erfolgreich übersprungen. Ein Beispiel für *Siedler5* ist bei „*Beispiele/Optimierer/Siedler5*“ und für das *griechische Festland* „Beispiele/Lauf/Griechisches Festland“ zu sehen.
* Die *spitzen Ecken* von *Siedler2* und *Siedler4* sind an sich vielversprechende Punkte für das Setzen der Ortschaften. Jedoch ist der hier verfolgte Lösungsansatz **generisch** gehalten. Er besitzt also *keine Sonderlogik*, um solche Ecken zu erkennen und zu berücksichtigen. Als Folge davon nutzen die erstellten Besiedlungspläne den Platz um die spitzen Ecken nicht immer voll aus.
* Die *Zufälligkeit* des *Laufs* und *Optimierers* wird z.B. in *Siedler2* deutlich. Dort wären die Besiedlungspläne ab *82* Ortschaften auch für nur *1* Gesundheitszentrum möglich gewesen. Auch bei anderen Plänen ist also davon auszugehen, dass minimal bessere Besiedlungspläne bei erneutem Durchlauf gefunden werden könnten. Dennoch ist das **Optimierungsergebnis** in der Hinsicht **robust**, dass der Zufall nur einen kleinen Einfluss auf die Zahl der erreichten Ortschaften hat (±2 bei *Siedler2*).

Der Schwierigkeitsgrad der Optimierung steigt mit der Zahl der Ortschaften. Wenn die Optimierung zu schwierig wird, sorgt die Budgetierung des *Laufs* und des *Optimierers* dafür, dass ein Gesundheitszentrum hinzugefügt oder der Lauf beendet wird. Dies wird im folgenden Schaubild veranschaulicht. Als Schwierigkeitsgrad wird hierbei die Zahl der *Optimierer-Iterationen* genommen, die zwischen zwei *zulässigen* Besiedlungsplänen benötigt wurden. Dies berücksichtigt sowohl die Neuplatzierungen der zusätzlichen Ortschaft als auch die nachfolgenden Punktbewegungen des Optimierers. Es werden jeweils die Stammfunktionen gezeigt, da die Verläufe sich sonst oft überschneiden würden. Die *Steigung* der Verläufe zeigt also den *Schwierigkeitsgrad*.

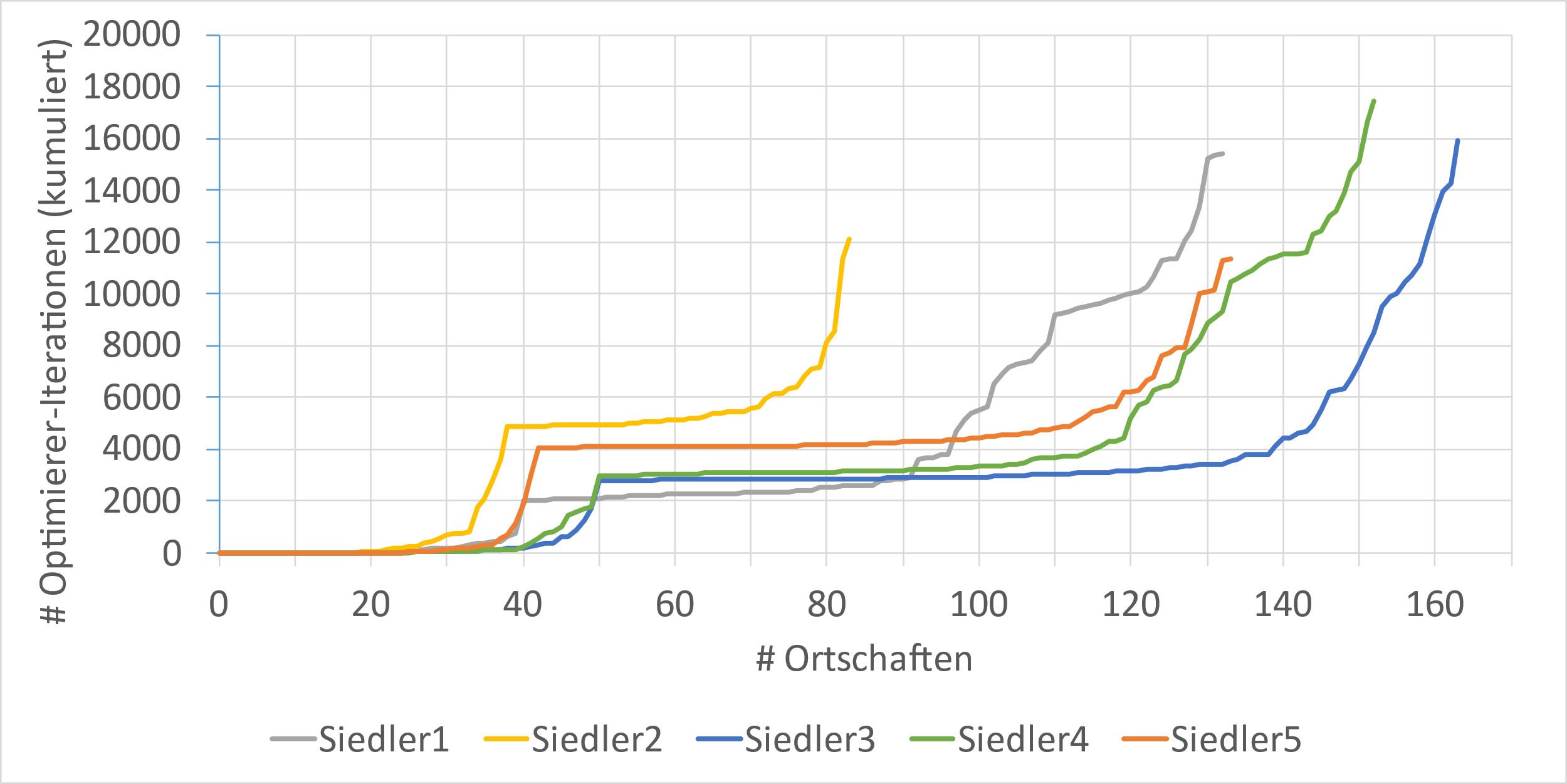


Abbildung 7: Benötigte Optimierer-Iterationen pro Ortschaften für die vorgegebenen Gebiete

Wie erwartet steigt der Schwierigkeitsgrad *vor* Hinzunahme des Gesundheitszentrums. *Danach* ist er wieder minimal, außer wenn keine zusätzliche Abdeckung durch das Gesundheitszentrum erzielt werden kann. Außerdem lässt sich erkennen, dass der erhöhte Schwierigkeitsgrad nicht abrupt einsetzt, sondern sich kontinuierlich aufbaut. Dies ergibt sich daraus, dass die Wahrscheinlichkeit, eine zulässige Lösung ohne Hinzunahme eines weiteren Gesundheitszentrums zu finden, stetig abnimmt.

## Zusätzliches Siedlungsgebiet

Auch für das *griechische Festland* als *realitätsnäheres* und *größeres* Siedlungsgebiet sind die Ergebnisse im Folgenden festgehalten.

Die Laufzeit ist um den *Faktor* **21,6** länger als die längste Laufzeit der vorgegebenen Siedlungsgebiete (*2851 min* statt maximal *132 min*). Dies liegt unter dem Faktor, der aufgrund obiger *Komplexitätsbetrachtung* zu erwarten wäre. Im Einzelnen sind die *eingehenden Faktoren*:

* Ortschaften (*n*): Faktor **5,2** (*851* statt maximal *163*)
* Gesundheitszentren (*z*): Faktor **3,0** (*9 statt 3*)
* Gebietseckpunkte (*g*): Faktor **7,5** (*165* statt maximal *22*)

Die folgende Tabelle hält den Zusammenhang fest zwischen der Zahl der verfügbaren Gesundheitszentren und der Anzahl der Ortschaften, die damit zulässig platziert werden können. Da das Siedlungsgebiet größer ist, tritt erst bei knapp *10* Gesundheitszentren eine Sättigung der möglichen Ortschaftszahl ein. Daher wurden auch mehr Werte als bei den vorgegebenen Siedlungsplänen ermittelt.

| **# Gesundheitszentren** | **# Ortschaften** |
| --- | --- |
| 0 | 257 |
| 1 | 366 |
| 2 | 420 |
| 3 | 499 |
| 4 | 621 |
| 5 | 707 |
| 6 | 765 |
| 7 | 800 |
| 8 | 838 |
| 9 | 851 |

Tabelle 4: Maximal erreichte Anzahl an Ortschaften nach Anzahl an Zentren für das griechische Festland

Die zugehörigen Plankarten sind im Folgenden abgebildet.

A screenshot of a map

Description automatically generated

Abbildung 8: Plankarten für das griechische Festland: maximale Anzahl an Ortschaften für eine gegebene Anzahl an Zentren

Aus den Plankarten lassen sich folgende Erkenntnisse gewinnen:

* Die *konkaven* Elemente und der *stark zerklüftete* Küstenverlauf des Siedlungsgebiets werden durch den Lösungsansatz sehr gut *berücksichtigt*. Dies zeigt sich z.B. an der Ausnutzung der anspruchsvollen Halbinseln [Chalkidiki](https://de.wikipedia.org/wiki/Chalkidiki) und den [Peloponnes](https://de.wikipedia.org/wiki/Peloponnes).
* Die größere Siedlungsfläche zeigt allerdings *Schwächen* bei der Platzierung der *Gesundheitszentren* auf. Werden diese initial *schlecht* platziert (z.B. in [Thrakien](https://de.wikipedia.org/wiki/Thrakien_(geographische_Region_Griechenlands)) statt in Zentralgriechenland), kann selbst die Punktbewegung der Optimierung dies *nur zum Teil* korrigieren und führt lediglich zu *lokalen Optima*. Dass diese gefunden werden, liegt an der verfolgten Strategie bei der *Bewegung* der Gesundheitszentren. Immer *alle oder keine* zu bewegen, entfällt als Option, da sonst bei den bereits *gut platzierten* Gesundheitszentren sehr wahrscheinlich *Lücken* entstehen würden. Stattdessen berücksichtigt der Optimierer dedizierte Kandidaten, in denen nur genau ein zufälliges Gesundheitszentrum (und keine Ortschaft) bewegt wird.

Zum Abschluss kombiniert das folgende Schaubild für eine variable Anzahl von zulässig platzierten Ortschaften zwei Aspekte: Zum einen wird die Zahl der dafür benötigten Gesundheitszentren angegeben. Zum anderen wird der *Schwierigkeitsgrad* der Hinzunahme einer Ortschaft wie zuvor durch die Zahl der dafür notwendigen *Optimierer-Iterationen* genommen.

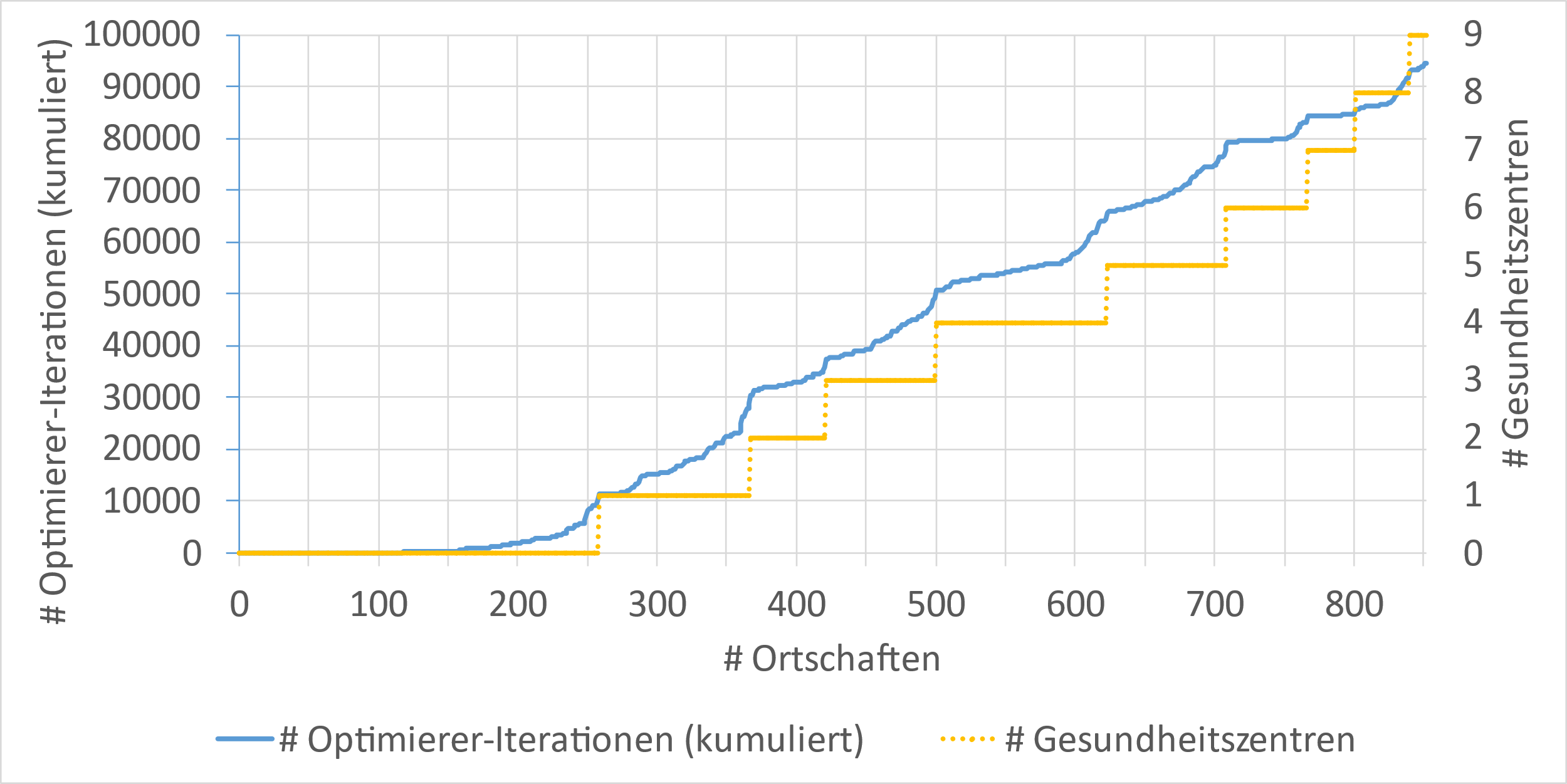


Abbildung 9: Benötigte Optimierer-Iterationen pro Ortschaften und den dafür benötigten Gesundheitszentren für das griechische Festland

# Beispiele

## Optimierer

Die folgenden Kartenreihen zeigen beispielhaft den Verlauf der Optimierung. Bei jeder *Iteration* werden mögliche Kandidaten durch *Punktverschiebung* geschaffen und anhand ihres *Loss* bewertet.

Die lila Punkte sind Ortschaften, die von keinem Gesundheitszentrum abgedeckt werden und zu nah aneinander liegen.

### *Siedler5* bei *128-130* Ortschaften und *1-2* Gesundheitszentren

Dieses Beispiel zeigt, wie Punkte (in diesem Fall das Gesundheitszentrum) erfolgreich *Gebietslücken* überwinden können. Dazu führt im Beispiel die (zufällige) Platzierung der Ortschaften Nummer *129* links unten und *130* links oben.

A graph of a function

Description automatically generated with medium confidence

Abbildung 10: Plankarten zu Siedler5 bei 128-130 Ortschaften und 1-2 Gesundheitszentren

### *Griechisches Festland* bei *813* Ortschaften und *8* Gesundheitszentren

Dieses Beispiel zeigt, wie die Verschiebung eines Gesundheitszentrums über den [Argolischen Golf](https://de.wikipedia.org/wiki/Argolischer_Golf) hinweg zur Abdeckung einiger ungeschützter Ortschaften führt.

A screenshot of a map

Description automatically generated

Abbildung 11: Plankarten zum griechischen Festland bei 813 Ortschaften und 8 Gesundheitszentren

## Lauf

Die folgenden Kartenreihen zeigen beispielhaft den schrittweisen Aufbau des Besiedlungsplans im *Lauf*. Bei jeder *Iteration* wird eine zusätzliche Ortschaft hinzugefügt und der Plan wird (soweit möglich) durch *Optimierung* zulässig.

### *Siedler 1*

Der Lauf endet bei Iteration *132*. Dieser letzte Plan ist im Feld für *140* dargestellt.

A screenshot of a graph

Description automatically generated

Abbildung 12: Plankarten zu Siedler1

### Griechisches Festland

Der Lauf endet bei Iteration *851*. Dieser letzte Plan ist im Feld für *860* dargestellt.

A screenshot of a map

Description automatically generated

Abbildung 13: Plankarten zum griechischen Festland (von Iteration 0 bis 480)

A screenshot of a map

Description automatically generated

Abbildung 14: Plankarten zum griechischen Festland (von Iteration 500 bis 851)

# Quellcode

Da der komplette Quellcode sehr umfangreich ist, werden im Folgenden nur Auszüge aus den Schlüsselpassagen der Lösung gegeben.

## Gebiet: ist\_drin

def ist\_drin(self, p: TPunkt):

"""

Implementiert die Strahlmethode (von links) fuer den Punkt-in-Polygon-Test.

Alle Schnittpunkte des Polygons mit einem horizontalen Strahl von aussen bis zum

Punkt werden gesammelt. Ist ihre Zahl ungerade, so ist der Punkt drin.

Um Probleme zu umgehen bei Linienzugspunkten (Lx/Ly) mit Ly == Py,

werden sie so behandelt, als ob ihre y-Koordinate Ly - ε betraegt:

Also Ly - ε < Py, was aequivalent zu Ly <= Py ist

Diese Methode wird zum Beispiel hier aufgefuehrt

"... considering vertices on the ray as lying slightly above the ray."

(https://en.wikipedia.org/wiki/Point\_in\_polygon#Limited\_precision)

Die gegebenen Gebiete werden in diesem Sinne interpretiert, d.h. Punkte am oberen Rand

des Gebiets gehoeren zu ihm, waehrend die auf dem unteren Rand dies nicht tun.

"""

px, py = p

war\_groesser = self.\_\_linienzug[0][1] > py

anzahl\_schnittpunkte = 0

for (lvx, lvy), (lx, ly) in it.pairwise(self.\_\_linienzug):

if war\_groesser != (ly > py):

war\_groesser = not war\_groesser

if ly == py:

sx = lx

else:

sx = lvx + (lx - lvx) \* (py - lvy) / (ly - lvy)

if sx < px:

anzahl\_schnittpunkte += 1

return anzahl\_schnittpunkte % 2 == 1

## Besiedlungsplan: auswerte\_ort\_abstand

def \_\_auswerte\_ort\_abstand(self):

zunahe\_orte = set()

# Einteilung in Planquadrate

planquadrat = {}

for x, y in self.\_\_orte:

px = x // self.param.sicher\_abstand\_ab

py = y // self.param.sicher\_abstand\_ab

p = px, py

if p not in planquadrat:

planquadrat[p] = []

planquadrat[p].append((x, y))

# Vergleich der Punkte eines jeden Planquadrats mit sich selbst

# und mit denen der Planquadrate rechts, linksunten, unten, rechtsunten

# Dies reicht aus, da

# - sonst keine zu geringen Abstaende vorhanden sein koennen

# - die Vergleich in eine Richtung fuer jedes Punktepaar benoetigt wird

# (also kein Vergleich mit Planquadraten, die links oder zum Teil oben sind)

kombinationen = []

for (px, py), orte in planquadrat.items():

kombinationen.extend(it.combinations(orte, 2))

kombinationen.extend([

(o1, o2)

for o1 in orte

for p2 in [(px + 1, py), (px, py + 1), (px + 1, py + 1), (px - 1, py + 1)]

for o2 in (planquadrat[p2] if p2 in planquadrat else [])

])

for o1, o2 in kombinationen:

erlaubter\_abstand = self.param.min\_abstand \

if {o1, o2} & self.\_\_geschuetzte\_orte \

else self.param.sicher\_abstand\_ab

abstand = self.\_\_berechne\_abstand(o1, o2)

if erlaubter\_abstand > abstand:

self.\_\_loss += erlaubter\_abstand - abstand

zunahe\_orte.update([o1, o2])

self.\_\_zunahe\_orte = frozenset(zunahe\_orte)

## GierigOptimierer: tue

BUDGET\_MAX\_ITER = 100

BUDGET\_MAX\_ITER\_OHNE\_FORTSCHRITT = 10

ZAHL\_KANDIDAT\_PLAENE\_BEWEGE\_ZENTRUM\_ALLE\_ORTE = 30

ZAHL\_KANDIDAT\_PLAENE\_BEWEGE\_ZENTRUM = 20

ZAHL\_KANDIDAT\_PLAENE\_BEWEGE\_UNGUELTIG = 10

def tue(self) -> tuple[int, Besiedlungsplan]:

iter\_num = 0

plan = self.\_\_plan

for b in self.\_\_beobachter:

b.optimierer\_start(plan, self.STRATEGIE)

iter\_num\_ohne\_fortschritt = 0

while iter\_num\_ohne\_fortschritt < self.BUDGET\_MAX\_ITER\_OHNE\_FORTSCHRITT \

and iter\_num < self.BUDGET\_MAX\_ITER and plan.hole\_loss() > 0:

iter\_num += 1

iter\_num\_ohne\_fortschritt += 1

kandidaten = {

self.\_\_erstelle\_kandidat(plan, mit\_zentrum=True, mit\_gueltigen=True)

for \_ in range(self.ZAHL\_KANDIDAT\_PLAENE\_BEWEGE\_ZENTRUM\_ALLE\_ORTE)

} | {

self.\_\_erstelle\_kandidat(plan, mit\_zentrum=True, mit\_ungueltigen=False)

for \_ in range(self.ZAHL\_KANDIDAT\_PLAENE\_BEWEGE\_ZENTRUM)

} | {

self.\_\_erstelle\_kandidat(plan)

for \_ in range(self.ZAHL\_KANDIDAT\_PLAENE\_BEWEGE\_UNGUELTIG)

}

# gierige Wahl: Fahre mit Kandidaten fort, der geringsten Loss hat

bester\_kandidat = min(kandidaten)

if bester\_kandidat.hole\_loss() < plan.hole\_loss():

plan = bester\_kandidat

iter\_num\_ohne\_fortschritt = 0

for b in self.\_\_beobachter:

b.optimierer\_iteration(plan)

for b in self.\_\_beobachter:

b.optimierer\_ende()

return iter\_num, plan